2

RADC-TR-90-406
Final Technical Report
December 1990

MULTILEVEL SECURITY FOR KNOWLEDGE BASED SYSTEMS

SRI International

Teresa F. Lunt and Thomas D. Garvey

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RADC-TR-90-406 has been reviewed and is approved for publication.

APPROVED: [illegible]

JOSEPH V. GIORDANO
Project Engineer

APPROVED: [illegible]

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: [illegible]

RONALD S. RAPOSO
Directorate of Plans & Programs

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE December 1990	3. REPORT TYPE AND DATES COVERED Final - -

4. TITLE AND SUBTITLE MULTILEVEL SECURITY FOR KNOWLEDGE BASED SYSTEMS	5. FUNDING NUMBERS C - F30602-87-D-0094, Task 6 PE - 35167G PR - 1068 TA - QC WU - 06
6. AUTHOR(S) Teresa F. Lunt and Thomas D. Garvey	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SRI International 333 Ravenswood Ave Menlo Park CA 94025	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Air Development Center (COAC) Griffiss AFB NY 13441-5700	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RADC-TR-90-406

11. SUPPLEMENTARY NOTES
RADC Project Engineer: Joseph V. Giordano/COAC/(315) 330-2925
The prime contractor for this effort is IIT Research Instituta, Maryland Technology Center, Lanham, Maryland.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

We discuss work aimed at defining a multilevel, mandatory security policy for knowledge based systems. We address two distinct issues: an effective implementation formalism based on a multilevel, object oriented programming paradigm, and requirements for ensuring the correctness of inferences computed on the basis of possibly contradictory information from different access classes. We define requirements for an object-oriented system capable of handling multilevel objects within a single access class. We then outline a method by which multilevel objects may be used to implement a simple knowledge based system built on production rules. We argue that the issues regarding correctness are similar to those of truth maintenance in standard knowledge based systems and may be addressed by similar methods.

14. SUBJECT TERMS Multilevel Security, Knowledge Based Systems, Computer Security, Object-Oriented Systems	15. NUMBER OF PAGES 64
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Contents

Chapter 1

Introduction and Overview

1.1 Introduction

The emerging technology of knowledge based systems (KBSs) is of great import for applications in business, industry, and the military. Many of these applications will involve the use of information with different degrees of sensitivity by users with different degrees of trustworthiness. Such applications require that information be segregated and managed in order to ensure that users can access only that information for which they are specifically cleared. To the best of our knowledge, however, none of the systems developed to date, under development, or in the planning stages explicitly addresses either mandatory or discretionary security. Although security is not a driving requirement of these systems, it seems unlikely that they will pass much beyond the proof-of-concept stage without the imposition of some security requirements. Thus these systems could not be used in a multilevel environment or in applications that require controlled access to portions of the system.

1.1.1 Overview of the Report

The work reported here addresses the problem of providing a multilevel secure framework for developing knowledge-based systems. We partitioned the problem into three stages:

- Definition of a security policy for object oriented programming systems (OOPS)
- Specification of a candidate knowledge based system (KBS) within an OOPS
- Investigation of issues of correctness within the KBS.

In addressing the first stage, we developed three distinct security policy models. These are referred to as the Multilevel Object Model, the Single-Level Object Model (also called the Millen-Lunt model), and the Composite Object Model. In the Multilevel Object Model, objects are truly multilevel, with components that range over different access cla~~r~~ses. The object itself is at an access class equal to or lower than the class of the lowest component or value it contains. In the Single-Level model, objects and their components are all at a single access class, but they may contain the names of objects at higher or lower access classes. In the Composite Object model, objects may be composed from other objects at equal or lower access classes only (as opposed to the Multilevel Object model). In this model, objects are also single-level, but the additional restriction preventing access to objects at higher access classes greatly simplifies the resulting security semantics and subsequent implementations. We recommend that the Composite Object model be the focus for future development and realization.

One of the goals of our security policy is to provide a framework that is habitable and supports the development of knowledge based systems without subjecting the developer or user to onerous restrictions or requirements. Furthermore, the simpler the policy, the easier it is to validate an implementation and to assure its effectiveness. This in turn makes it easier to provide high levels of assured security.

The remainder of the report focuses on the Composite Object model, the first two models having been reported on in our Interim Report [15]. We begin with a brief overview of background material, including a description of the Multilevel Object model and the Single-Level Object model. We then describe the Composite Object model in detail. Following that, we describe a notional KBS based on production rules that could be constructed within the OOPS environment. We describe its operation using a detailed example. We conclude the report with a detailed comparison of the models and recommendations for future work.

1.1.2 Objectives of the Approach

In this work, we were motivated by the desire to develop a security policy that would be simple to describe and implement, but that would not prove overly constraining to system developers. The perspective we adopted was that of a hierarchically-distributed system. That is, from the user's perspective, the data or knowledge base is distributed across a hierarchy of program/database partitions at and below the viewer's apical position. A lower level user will have a view of the data/knowledge base which includes only the part of the hierarchy at the user's level and below.

Our model is slightly more constraining than mandatory security requires in that we do not allow users to be aware of information stored at higher levels. Neither can they invoke procedures involving higher level objects or modify higher level data. Of course, users are not allowed to modify lower level data, but this is dictated by mandatory security. This additional constraint does not seem to unduly limit a system developer, and it has the advantage of leading to a security policy that is extremely simple to enforce. In our model, security is inherent in the structure of the programming architecture, as the program structure mirrors the security structure; security does not require treatment of the special cases that arise when low-level subjects are allowed to invoke high-level procedures, such as how to handle returned values. Our model requires only a reference monitor to enforce mandatory security.

We assign security levels, to processes or subjects, derived from the clearance of the user on whose behalf the subject is operating. When a user invokes the KBS, an instance of the system runs as a subject with an access class equal to the user's clearance. Hence the only objects available to that subject are those visible at (i.e., dominated by) the subject's access class.

This approach implies that all functions must be carried out by single-level subjects. The use of only single-level subjects for routine processing provides the greatest degree of security possible, and considerably reduces the risk of disclosure of sensitive information. Thus, the KBS, when operating on behalf of a user, cannot gain access to any data whose classification is not within the user's clearance.

This implies a KBS design that does not rely on a single server to service all users. On the contrary, the system must support multiple server instances that share the same logical knowledge base.

Our model segregates security issues from programming issues; the system developer does not need to be concerned with security, except for one point: because of multilevel security, the information stored in one partition may contradict that stored in another, and the different partitions may not have access to the same information. This means that the user must verify results computed from information stored at lower levels in the hierarchy. This raises issues of truth maintenance that must be addressed by system implementers.

Chapter 2

General Background

Before describing our policy for a multilevel knowledge based system, we review several key concepts. We first describe the concept of multilevel security. Next, we provide a brief overview of object oriented programming concepts. In Chapter 4, we discuss production systems and frame representations, two representative methods used in a KBS.

2.1 Overview of Multilevel Security

The concern for multilevel security arises when a computer system contains information with a variety of classifications and has users who are not all cleared for the highest classification of data contained in the system. The *classification* of the information to be protected is defined as the potential damage that could result from unauthorized disclosure of the information. The *clearance* assigned to a user is defined as the user's trustworthiness to not disclose sensitive information to individuals not cleared and thus not so trusted. We use the term *access class* to include both user clearances and information classification.

2.1.1 The Multilevel Security Lattice

An *access class* consists of a hierarchical sensitivity level (e.g., TOP-SECRET, SECRET, CONFIDENTIAL, UNCLASSIFIED) and a set of nonhierarchical categories. In order for a user to be granted access to classified information, the user must be cleared for the hierarchical sensitivity level as well as for each of the categories in the information's access class. For example, information might be classified TOP-SECRET SPOOK OUTER-SPACE. In order for a user to obtain access to this information, the user would have to be cleared for TOP-SECRET as well as for the categories SPOOK and OUTER-SPACE. The sensitivity levels are hierarchical because they are linearly ordered, e.g., TOP-SECRET is greater than SECRET is greater than CONFIDENTIAL is greater than UNCLASSIFIED. Thus, for example, persons cleared for TOP-SECRET information may also have access to SECRET information. The categories are nonhierarchical because they do not have such a linear ordering. However, the set of access classes (<sensitivity level, category set> pairs) is partially ordered and forms a lattice [4]. By a *partial ordering* we mean that given any two access classes, either one is greater than or equal to the other, or the two are said to be noncomparable. For example, TOP-SECRET is greater than SECRET, but TOP-SECRET SPOOK and TOP-SECRET OUTER-SPACE are noncomparable, because neither is greater than the other: a TOP-SECRET SPOOK user cannot obtain access to TOP-SECRET OUTER-SPACE information, and vice versa.

We call the partial ordering relation on the lattice of access classes the *dominance* relation. We say that one access class A dominates another access class B if the sensitivity level of A is greater than or equal to the sensitivity level of B and if the security categories of A include all those of B. For example, TOP-SECRET SPOOK OUTER-SPACE dominates TOP-SECRET OUTER-SPACE.

In a multilevel computer system, we use *system-low* to refer to that access class handled by the system that is dominated by all other access classes handled by the system; we use the term *system-high* obversely.

2.1.2 Mandatory Security

The Department of Defense (DoD) policies restricting access to classified information to cleared personnel are called *mandatory* security policies. In addition to those required for mandatory security, additional access controls may be imposed by a site or system. These additional controls enforce what is called *discretionary* security. The access controls commonly found in most operating systems are examples of discretionary access controls. There are many discretionary security issues surrounding any KBS; for example, different types of users (engineers, managers, maintainers, testers, and consumer-users) may have different access rights to the rules and data in the system. However, although we believe discretionary security concerns are significant and merit serious attention, for the purposes of this paper we are concerned only with mandatory security.

Mandatory security requires that classified data be protected not only from direct access by unauthorized users, but also from disclosure through indirect means, such as covert signaling channels and inference. *Covert channels* are channels that were not designed to be used for information flow but that can nevertheless be exploited by malicious software to signal high classification data to low clearance users. For example, a high process, a program instance with a high access class because it is acting on behalf of a high user, in order, to encode high information, may use read and write locks observable to a low process over time (e.g., locked = 1, unlocked = 0). Inference occurs when a low user can infer high information based on observable system behavior. For example, a low user attempting to access a high object can infer something quite different according to whether the system responds with "object not found" or "permission denied."

Thus, mandatory security requires that no information can flow from high access classes to low. The mandatory access control requirements are formalized by two rules, the first of which protects data from unauthorized disclosure, and the second of which protects data from contamination.

- *Simple Security Property*: A subject S is not allowed to read data of access class c unless $classification(S) \geq c$
- $\star$*Property*: A subject S is not allowed to write data of access class c unless $classification(S) \leq c$

In the above rules, a *subject* is a process acting on a user's behalf; a process has a clearance level derived from that of the user.

The simple security property and ⋆property above are based on properties of the same names defined in the Bell and LaPadula security model [1]. The ⋆property is intended to prevent sensitive information from being transferred to an object whose access class is not dominated by that of the information, so that the information will not become accessible to users who are not cleared for it. This confinement property prevents subjects, represented by software acting on behalf of users, from writing down (including writing to noncomparable access classes). The motivation for the ⋆property is that whereas cleared personnel can be trusted not to disclose classified information deliberately, computer programs, through errors of design or implementation, might move classified information to a location not protected at the required access class, or might contain Trojan horses that deliberately and maliciously violate security. The ⋆property is said to enforce confinement, since the effects of a malicious program are confined to objects at the subject class and higher. A somewhat weaker property would allow a subject to write down when the subject does not have read access to information whose classification dominates the classification of the information being written. Although this weaker property would prevent malicious software from copying down, its enforcement would be more complicated than for the relatively simple ⋆property.

Mandatory security intends to prevent users from drawing inferences from the perceived presence, appearance, and disappearance of objects; prevent the observable behavior of the system from depending on information classified higher than the user's clearance; and prevent users' visible effects from being used as a signaling path.

2.1.3 The Reference Monitor

To satisfy mandatory security in multilevel computer systems, we assign access classes to processes, derived from the clearance of the user on whose behalf the process is operating. Traditional practice is to segregate the security-relevant functions into a *security kernel* or *reference monitor*. The reference monitor mediates each reference to an object by any process, allowing or denying the access based on a comparison of the access classes associated with the process and with the object. The reference monitor must be tamperproof; it must be invoked for every reference; and it must be small enough to be subject to analysis and test that can be assured as complete. When assurance is important, the reference monitor is subject to *formal analysis*; that is, formal mathematical proof, sometimes by using automated tools, that the monitor correctly enforces the mandatory security policy. The result of such analysis is called *formal verification*. The reference monitor forms the core of the trusted computing base (TCB), which contains all security-critical code.

The DoD has developed evaluation criteria for trusted computer systems [6]. These criteria incorporate the concept of the reference monitor and include requirements for assurance as well as many other security requirements. The "trust" in trusted computer systems rests on the ability to provide convincing arguments or proofs that the security mechanisms work as advertised and cannot be disabled or subverted. These criteria include requirements for "minimizing the complexity of the TCB, and excluding from the TCB modules that are not protection-critical," so that the reference monitor is "small enough to be verifiable" [6]. Without such a requirement, a high degree of assurance would not be feasible.

The approach we take here assumes an existing verified reference monitor underlying and constraining the KBS; thus, the KBS itself does not enforce any portion of the mandatory security policy. This is similar to the approach taken for the SeaView multilevel database system [14] and is consistent with the general security architecture known as TCB subsets [18].

2.2 Object Oriented Programming Systems

It is often convenient to organize programs around objects, which model real-world entities.[1] Each object has some state and a set of operations that can be performed on it. An object's state is represented by a set of instance variables which are part of the object definition. Operations on objects are handled by methods, which are executed in the context of the object's state upon the receipt of messages. Object oriented programming (OOP) is a powerful technique for organizing and managing very large programs, which

[1]Much of the material in this section came from [15].

would otherwise be impossibly complex.

An OOP consists of a set of objects and a set of operations on these objects. The design of such a program consists of three major tasks:

- Choosing the kinds of objects to provide in the program
- Defining the characteristics of each object
- Determining the operations to perform on each object.

2.2.1 Objects

OOP, as originated in the Smalltalk system [8], adopted the idea of a class hierarchy and carried it further by introducing message passing and methods. An *object* represents either a class or an individual, and stores named attributes *instance variables.*

Smalltalk supports both a subclass and an instance relation, wherein an instance object is an individual, by which relation it can have no instances of its own. Both subclasses and instances inherit the variables of the parent class object, and may have additional variables of their own.

For the purposes of this report, we will not make use of the distinction between subclass objects and individual objects. We will use the term "instance" ambiguously to refer either to a subclass or to an individual instance. The difference between a subclass and an individual instance is significant primarily for reasons of implementation efficiency; it does not affect security policy. Thus, our model is equally valid for systems that make this distinction and those that do not.

2.2.2 Methods

An object has methods defined for it. Methods encapsulate the behavior of an object, in that an object can be acted upon only through executing the methods defined for the object. Methods are invoked by sending messages to an object. A message consists of a command, which selects the appropriate method, and some arguments, if necessary; this is illustrated in Figure 2.1.

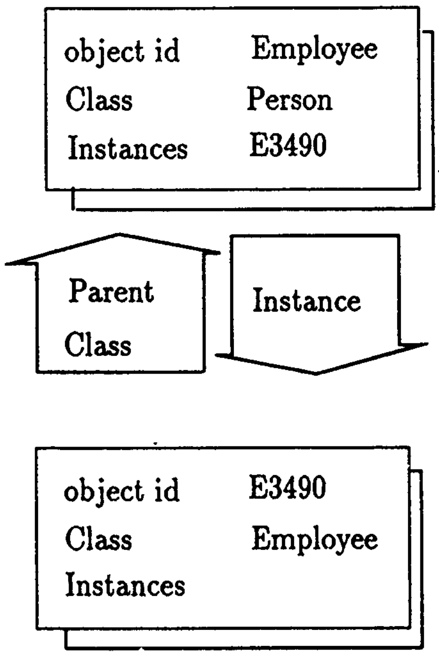

Figure 2.1: Message Processing

A method performs three sorts of activities: it may read and write the variables of the object where it resides; it may send messages to other objects, to invoke methods there; and it may, when it terminates, return a value to the sender of the message that invoked it. The transmittal ability provides for much of both the usefulness and the complexity of object oriented systems.

For example, a bank-account object would have methods for the commands Withdraw, Deposit, and Query-balance. To withdraw or deposit funds in a specific bank account, a message of the form *command value* is sent to the object representing that bank account, where *command* is either Withdraw or Deposit, and *value* is the amount to be deposited or withdrawn. To query the balance for a specific account, the message Query-balance would be sent to the account object.

Methods are inherited by instances. This means that a method placed in a class object is automatically made available to its descendants. The Withdraw method for bank accounts, for example, need only be placed in the Bank-account class object, to permit all individual bank accounts to respond to a Withdraw message. Inheritance of methods is one of the greatest benefits of the object oriented approach.

2.2.3 References to Objects

Every object has a unique identifier called an *object id.* A reference to an object is needed when one wishes to send a message to it; the reference is the object's address. Here, we will treat the object id as the name of the object (and vice versa). An instance variable will be identified by the name of its containing object conjoined with its own when necessary for unambiguous reference. Our multilevel security policy will involve interactions among objects in different address spaces. In these cases, an object id will be unambiguously specified as a *pathname*, consisting of the address-space name conjoined with the object name.

Objects can be linked to one another by references in suitable variables. References can represent any of a multitude of relationships. For example, an Employee object may have a Job variable. Its value is often not just the name of a job, but rather a reference to another object containing information about the job, such as its duties and salary range.

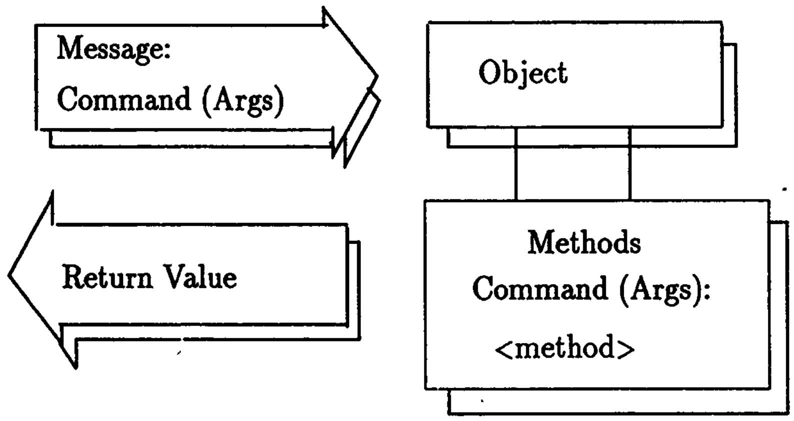

Figure 2.2: The Class-Instance Relation

Objects are linked also by the class-instance relation. In Smalltalk, this relation is implicit and maintained by the system. For the purposes of modeling security in object oriented systems, we shall make this relation explicit, by assuming the existence of standard variables called "Class" and "Instances" in every object. The value of the Class variable is the identifier of the parent class of the object. The value of the Instances variable is a (possibly empty) list of identifiers of the object's instances. Figure 2.2 illustrates the class-instance relation.

2.3 Objects as the Basis for Knowledge Based Systems

The OOP model is a natural extension of the frame-based model for knowledge [16] and is a natural formalism for constructing a KBS. Object oriented models support the notion of a class hierarchy, an essential feature of a KBS. In an earlier report, we discussed how the object model can easily support a KBS based on production rules [15]. Furthermore, virtually every KBS is based on a logical foundation of some kind. We believe that most of these can also be supported by the framework described in this report. In particular, systems for reasoning with uncertain information tend to provide a propositional framework for representing the space of possible events. In classical probability theory [9] and in nonclassical methodologies such as the Shafer-Dempster Theory of Evidence [17], for example, primitive events are represented as a set of mutually exclusive and exhaustive statements. These event spaces could be represented adequately as objects.

2.4 Security Policy for Object Oriented Programming Systems

For the past year we have been conducting an investigation into a model of security for a multilevel KBS. Because our initial investigations have shown that a KBS can be straightforwardly modeled and implemented on object oriented systems, much of our work has focused on developing a security model for an object oriented database system (OODBS).

2.4.1 Multilevel Object Model

Our first model of a secure OODBS was developed under the SeaView project [12]. In this model, classifications can be associated with objects (including classes) and facets of objects, where a *facet* could be an instance variable, a method, or a constraint. Thus, in this model, objects can be multilevel; that is, various portions of an object, such as class name, instance variable names, and methods, can have different classifications.

We defined a *hierarchy property*, which requires that the access class of an object must dominate that of its parent class object. This property is needed to permit the object to inherit methods and variables from its parent. This is a fundamental property that will play a part in any model of security for object oriented systems.

This model also requires that an object's facets be classified at least as high as the object itself. This is analogous to the SeaView property that requires that any data contained in a tuple be classified at least as high as the tuple's primary key, and that any data that can be stored in a relation be classified at least as high as the name of the relation [13]. We called this the *facet property* for our multilevel object model.

2.4.2 Single-Level Object Model

The semantics of the Multilevel Object model are relatively complex, and would lead to significant complexity in any implementation. For this reason, we turned to a security policy based on single-level objects (discussed in detail in [15]). Here, although the objects themselves are at a single access class, they may contain references (such as object identifiers) to objects at other access classes, both higher and lower. These references do not have security classes of their own, but are at the class of the object.

Subjects operating within an access class may send messages to higher objects, resulting in the creation of a subject at the higher access class. Values may not be returned from these higher subjects, in order to avoid the problem of writing down.

In the Single-Level model, the access class of objects that are created must dominate that of the subject that requested the creation. This enables low subjects to create high objects.

This single-level object model includes the hierarchy property, discussed above. We demonstrated that typical database security and integrity policies can be supported by this model.

We felt that the requirements that were imposed by allowing low subjects to access high objects were still too complicated for straightforward implementation and verification. In order to simplify such requirements and still provide effective multilevel security, we introduced the Composite Object model with the additional restriction of forbidding access to high objects by low subjects. In the remainder of the report, we discuss this model in detail.

For our current project, we developed a security model for an OODBS in which objects are single-level rather than multilevel [15]. The reasons for doing this were twofold: first, several difficulties with multilevel objects convinced us that we needed a model for single-level objects to put us on a more solid footing; secondly, we felt that a single-level object model might be sufficient to support a multilevel KBS.

This single-level object model includes the hierarchy property, discussed above. We have demonstrated that typical database security and integrity policies can be supported by this model [15].

2.4.3 Composite Object Model

More recently, we have developed a new security model that allows one to represent multilevel entities without the difficulties encountered in the multilevel object model. (See [12] for a discussion of these difficulties.) This model also retains the simplicity of the single-level object model. We present this new composite object model in the next chapter.

Chapter 3

The Composite Object Model

Here we present a composite object model that supports multilevel objects. Our considerations of typical intelligent applications have led us to the conclusion that multilevel objects are desirable for a KBS. Our composite benefits from the advances of, and avoids most of the complexities of, the initial multilevel object model developed under the SeaView project, and retains the simplicity of the Millen-Lunt single-level model. In our new composite object model, the access class of the object dominates the access class of its components (as contrasted with the facet property of the multilevel object model [12]. Each of these components is itself an object, which in turn can be multilevel and can consist of lower level components. This provides a very simple way of constructing multilevel objects. This model also leads to a natural decomposition of multilevel objects from single-level partitions. Thus, queries and updates on multilevel objects can be decomposed into queries and updates on the single-level partitions. This approach is easily generalized to an object model in which the object base is distributed and queries and updates are distributed across numerous processors and memories.

3.1 General Properties of the Composite Object Model

Because of the lack of a consistently applied semantic interpretation, associating labels with the various structural elements in a database system involves certain ambiguities. In order to define the meaning of an object being classified, we must specify exactly what information we intend to protect when we assign an access class to an object.

Three dimensions of classification have been proposed [19]:

- The data itself may be classified.
- The existence of the data may be classified.
- The reason, or rule, for classifying the data may be classified.

Our model does not contain explicit rules for classifying information. Thus, we do not need to address the third security dimension.

In addition, in our model, we collapse the first and second security dimensions. Thus, in our model, when we say that an object has access class c we mean that the information it contains has classification c and that the fact that the object exists is classified c. This means that in our model, the existence of an object should be hidden from lower level subjects.

The existence of an object can be inferred by the appearance of its name or object id as the value of a variable in some other object. Thus, an object id for a high object cannot be stored in a low object. In other words, low objects cannot contain references to high objects. We feel that this is a reasonable restriction, because in our model all multilevel objects are composite objects, wherein the access class of the composite dominates that of its components. Thus, the composite object can point to its components, but not vice versa. Traversals of the object hierarchy can reasonably be expected to move from composite objects to component objects.

This also means that the creation of composite objects must begin with the creation of the components. To create a composite object, a subject creates an object, then builds references to the already existing components. There is no need for backward references.

One consequence of this choice of security semantics is that if a subject creates a high object out of or from a lower class or type, then a reference to the new high offspring cannot be stored in the low parent object. So that high processes can traverse the object hierarchy from top to bottom, the system might include an object index at each access class that lists objects and their parents. Entries to these indexes would be made as a side effect of the "create object" operation. Another alternative would be to allow the creating subject to decide where to store the id of the new object. It must be stored in an object of the same access class as the creating subject, and it must be stored somewhere, or else the new object will be inaccessible.

The attributes of our security model are summarized according to the properties enumerated in the following section.

3.2 Security Properties of the Composite Object Model

In the security policy proposed here, we allow multilevel objects, or objects with facets from multiple access classes to reside in the same program partition. The classification of the object dominates both the name of the facet and its value. This is captured in the Object Dominance Property.

Property 1 (Object Dominance Property). The classification of an object dominates that of its facets. The facets may have different classification levels. □

We also require that the existence of an object have the same classification as the object itself. Therefore, low objects should not be aware of high objects.

Property 2 (Object Existence Property). The classification of the existence of the object, i.e., the classification of the object id, is the same as the classification of the object itself. □

This is handled in our proposed implementation by the fact that object ids are pathnames relative to the user's partition, and there is no means to refer to a higher partition.

The access class of an object is constrained by the following property.

Property 3 (Hierarchy Property). The access class of an object must dominate that of its parent object. □

This property is needed to permit the object to inherit methods and variables from its parent. Any attempt to read an object by reading one of its variables, or to execute it by sending a message to it, may implicitly read the object's parent class and perhaps the parent's parent class, and so on, until a default value or appropriate method is found.

We implement the security policy by creating a program lattice that mirrors the security lattice. Each access class in the security lattice has a corresponding partition in the program lattice. The following properties formalize this relation.

Property 4 (Correspondence Property). The program lattice is isomorphic to the security lattice. There is a one-to-one correspondence between partitions in the program lattice and access classes in the security lattice. □

The highest partition will correspond to the system-high access class. Just as a particular access class dominates all inferior access classes, a particular rtition will dominate all inferior partitions.

Property 5 (Partition Dominance Property). A partition in the program lattice corresponding to access class S dominates a partition corresponding to access class B iff A dominates B. □

All references to objects must be handled by the trusted reference monitor (RM). To enforce this rule, we require that each program partition exist in its own local address space; the only way to access information in different address spaces will be through a call to the RM.

The structure of the program lattice is maintained locally through a distinguished index object in each partition. The *index object* will contain (or be able to compute easily) the following information:

- A list of the partition's directly inferior partitions
- A list of all objects in the partition.
- A list of all object instances in the partition

The list of partitions will provide the means for traversing the lattice structure through recursive calls to the RM. The list of objects will enable the RM to quickly access components of particular objects in a partition. The list of instances will simplify the operation of constructing instances from objects in inferior partitions.

Objects and facets are identified by local pathnames which specify the partition and the object to the level of detail necessary for unambiguous identification.

We allow an object to read down, but in order to avoid illegal information flows, we do not allow it to write down. Our policy has the following read/write properties.

Property 6 (Read Property) A subject S can read an object O only if S dominates O. □

A subject can write an object only if its access class equals the access class of the object. This means that a subject cannot write down.

Property 7 (Write Property) A subject S can write an object O only if they have the same access class. □

An attempt to write down will be intercepted by the RM and one of two possible actions could occur: the RM could return an error, or it could create a polyinstantiated (or *shadowed*) object or facet in the partition of the subject.

Our write property differs from the traditional Bell and LaPadula model [1] in that in our model, subjects cannot write up, while in the Bell and LaPadula model they can. Furthermore, in our approach, an object in a particular partition will not be aware of higher level objects, nor will they be accessible to it.

3.3 Required Operations for the Composite Object Model

Within a multilevel security policy, we must address the issues of creation, modification and destruction of objects and instances, message passing and method evaluation, and reading and writing of instance variables.

3.3.1 Implementation of Partitions

We propose that partitions be implemented as separate program address spaces with specific access classes; access to these address spaces would be mediated by the RM. Each partition would maintain a list of subordinate partitions with locally unique names. References to these partition names would be mapped by the RM into its global partition table.

These partitions could comprise a permanent structure and represent partitions in an OODBS.

3.3.2 Object Composition

One of the key advantages of object oriented systems is the ability of one object to inherit properties from other component objects. In the security policy outlined here, an object at one access class may be composed of objects at the same or lower access classes. The access class of the new object will be the same as that of the creating subject. When a new object is defined, additional instance variables may be specified. These new variables, of necessity, will be classified at the access class of the new object because instance variables do not exist independent of objects, variables will either be inherited from pre-existing component objects, or will be specified when the object is defined. As in a standard OOPS, when instance variables with the same name are inherited from component objects, only one will be retained.

We do not differentiate between instances of an object class and the object class itself for inheritance. That is, object instances can be constructed from other object instances.

Newly defined instance variables may shadow those from other objects in the normal fashion. This will allow a high object, for example, to create its own version of variables that would otherwise be inherited from a low object.

Similarly, methods may be inherited from component objects. As with variables, methods may be shadowed in order to allow a high object to carry out computations differently than a low object, while using the same invocation sequence.

In an implementation of this model, we assume that every object and facet will be uniquely identified by a relative pathname which includes the name of the partition owning the object, the object name, and , where appropriate, the facet name.[1]

A low user may modify, redefine, or destroy an object at the low access class, even when that object is a component of a high-level object. The lower user will have no way of knowing of such high references or links to the low object. Therefore, we place the responsibility for ensuring the consistency of information with the high user. In order to facilitate this, we propose that each partition have an Error Handler that will take appropriate action when reference to a nonexistent object or variable is made.

If a stable copy of a low object or facet is needed, i.e., a copy that cannot be modified or deleted by low users, then a high copy should be created that polyinstantiates the low object. This will allow the high user both to guarantee the stability of the object's value and to monitor any changes low users might make, since the value of the low, polyinstantiated object will still be accessible through its unique pathname. This approach may not be desirable for situations where the low data is highly fluid and the high subject requires a stable information base.

3.3.3 Method Evaluation

In an OOPS, a method is invoked upon receipt of a message by an object. The message specifies the method and provides any arguments pertinent

[1]Since we assume these unique pathnames, we do not distinguish *composite* objects, where an object's components retain their identities, from *complex objects* that are created from other objects whose independent identities are lost in the creation process.

to the evaluation. This process is carried out by creating a subject at the access class of the object sending the message. The method is processed at this access class, and the RM enforces read and write properties based on this access class. The method should always return a value; the access class of the value returned will be that of the subject itself. This may result in different copies of the same information classified both low and high, as when the value of a low instance variable is returned to a high subject); in these cases, we recommend the use of an advisory label [14] indicating the access class of the origin of the information. Such a label is for information purposes only; it is not used in any release decision by the RM.

The use of *functional closure* [20] allows the message sender to specify the variable binding environment to be used during method execution. A functional closure encapsulates a process and a binding context in order to control the process execution environment. Specifically, by providing an association list of variable names and desired bindings in the message, the sender could effectively shadow variable bindings which would otherwise be used. For example, the sender could specify that the value for a low variable's value be used rather than the high value which would otherwise be used by default.

3.3.4 Reading and Writing Instance Variables

A composite object may inherit instance variables from lower level component objects. In order to ensure that read and write (and initialization) operations do not violate the security policy, we require that all such accesses be implemented via methods which will entail a call to the RM.

We propose to implement references to low variables from high objects by storing, in the value slot of the high instance variable, an indirect pointer to the variable in the low object. Until an attempt is made by the high-level object to write an instance variable inherited from the lower level, the lower-level variable will be accessed through this indirect pointer. If the high-level object attempts to write the variable, the reference monitor signals an error and the error handler will replace the indirect pointer with the new value (or, possibly, a pointer to the new value), which will effectively *shadow* the lower-level variable. An alternative method, if it is desirable to monitor the information stored in the low object, would be to add a new instance variable to the object in such a fashion that either the old or new values could be accessed under control of the high object, for example by adding it to the "front" of the list of instance variables (with respect to the order in which instance variables are accessed), rather than replace the indirect pointer; this would polyinstantiate the variable, but leave both high and low values accessible.

In this model, all updates would be single-level updates, and thus the model raises no new issues with regard to concurrency control or transaction management beyond those already under investigation in the database security community (see, for example, [7]).

Chapter 4

Multilevel Knowledge Based Systems

Here, we first provide a brief introduction to typical KBS programming constructs. We then discuss issues related to the implementation of a particular form of KBS, the production system, within an object oriented framework. Finally, we turn to issues of correctness within a hierarchical, multilevel KBS.

4.1 Knowledge Based Programming Constructs

The most prevalent knowledge based programming methodologies involve the use of rule based and/or frame based techniques. Here, we briefly review these concepts.

4.1.1 Production Rule Systems

One of the most ubiquitous forms of KBS is the production system (PS), where knowledge is represented in the form of production rules and declarative statements [2]. Informally, a rule may be described in the following format:

Rule 0 If <CIRCUMSTANCES>
then <DO ACTION, OR CONCLUDE SOMETHING>

A production system provides a knowledge base of such rules, working memory for the storage of facts and data representing the current state of the system, plus a rule interpreter that selects the next rule to be activated (or "fired"), that applies it to the working memory, and that carries out the resulting action.

In a production rule, the <circumstances> component is called the *antecedent* and the <do action, or conclude something> component is called the "hypothesis" when the action is to alter the working memory or the *action* when the result is to carry out some other process. In this paper, we shall refer to this component as the *hypothesis.*

There are two distinct processing paradigms that might be used in a PS. The first is called *forward chaining* (or *data directed processing*), which matches data in the working memory against the hypothesis components of each rule in order to produce a set of candidate rules. During the matching process, variables may be bound which may cause other rule components to be more precisely specified. A single rule from the candidate set is selected and activated which may possibly result in a modification of the working memory. This process is repeated until the desired result is obtained.

An alternative strategy is called *backward chaining* (or *goal directed* processing) which matches goals stored in the working memory against the antecedent component of each rule, in order to determine which rules might satisfy the goal. A single rule from the set of candidates is selected, and its hypothesis is posted in the working memory as a new goal. The process iterates until an antecedent subgoal is posted which is already in the working memory and, therefore, known to be true.

Mixed strategies are often employed which involve both goal directed and data directed processing.

4.1.2 Frame Systems

The class-hierarchy aspect of object oriented programming grew out of Minsky's "frame" idea for organizing knowledge [16]. In an artificial intelligence context, a frame represents a concept. More concretely, a frame defines the common attributes of a class of objects. An attribute is stored in a named slot. Defining an attribute says that the slot is present, and also, in some cases, gives the slot a default value.

Frames are hierarchically related according to the generality of the concepts they represent. If one frame represents a class of objects, frames below it in the hierarchy represent subclasses, not arbitrary subsets but more specific concepts.

Inheritance is essential to the frame idea. A slot in a class frame is inherited by its subclass frames. The fact that the class Project has Project-leader as a slot means that the class Research-project automatically has the Project-leader slot as well, and so will any subclasses of Research-project.

Given the intimate relationship between frames and OOPS, an implementation of a frame system as an object oriented system is quite natural. Therefore, we will not address such an implementation here.

4.2 Object Oriented Implementation

Here we describe the implementation of a simple production system (PS) within an OOPS.

4.2.1 Objects

There are numerous ways in which a PS may be implemented within an OOPS. Our approach will be to define the following classes of object.

- **Database object** for storing, searching, and accessing the working memory for the PS

- **Rule object** containing slots for the antecedent and hypothesis components of the rule, containing methods for determining whether the rule can be activated and containing methods activating the rule and returning the results of activation
- **Rule Base object** containing the set of rules
- **Pattern object** containing the patterns that compose the antecedent and hypothesis components of a rule as well as the statements in the working memory, and containing patterns with no variables or universally quantified variables *quantified pattern*
- **Binding object** consisting of a pattern plus a list of binding pairs which result from matching two pattern objects and contain variable names and values (which may, inturn, be a variable name)
- **Rule Interpreter object** containing slots for the current working memory and the current rule base, and including the necessary methods for selecting the next rule which could be activated, for resolving conflicts, for causing selected rules to be activated, and for initiating changes to the working memory

A database object will include slots for pointers to lower level database objects stored in subpartitions. Searches through the database, then, will continue until all lower-level databases have been searched. In general, the order of searching lower level databases will be determined by their order within the database object. This has the effect of making an object at one partition effectively range over all appropriate subpartitions.

This concept of hierarchical continuation will also be used for rule objects as well as for frames and other representations such as propositional sets used for tasks such as reasoning with uncertain information.

4.2.2 Operations

A prototypical forward-chaining processing sequence might follow these steps.

1. The user sends a START message to the current Rule Interpreter (RI) instance.

2. The RI cycles through its current rule set, sending MATCH-P messages to each of the rules.

3. Each rule, upon receiving the MATCH-P message, passes it to the antecedent pattern object. The pattern object attempts to match its pattern to each statement in the working memory. If a successful match is found, a binding object is created and returned for storage in the rule; otherwise, the match returns FALSE.

4. The RI compiles a list of those rule instances for which a successful match was found and stores them as the candidate rule set.

5. If there is more than one candidate rule, the RI invokes the conflict resolution procedure by sending a message to itself in order to select one rule for activation.

6. The selected rule is sent an ACTIVATE message, causing it to return either a new entry for the working memory, or a process to be executed.

7. The RI then either executes the specified process, or, by sending the ADD-DATA message to the database object, with the new data, the rule, and the antecedent binding object as arguments, it adds the new entry to the working memory, along with the basis for inferring the new data.

8. At the conclusion of the process, the RI may decide to terminate, or to send itself a CONTINUE message to continue the process.

At the conclusion of this process, new data items may be added to working memory. Each item will be qualified with the data and inference chain that led to the data.

A prototypical backward-chaining process might consist of these steps:

1. The user sends a SATISFY message to the RI object with the goal to be achieved as its argument.

2. Upon receipt of the SATISFY message, the RI examines the goal and determines whether the goal is a single statement, a conjunc[illegible]n, or a disjunction. A recursive approach is used for handling each case.

Single Goal: (continue the process) with no special action

Disjunction: select an evaluation order and map over the disjuncts, sending the SATISFY message to the RI with each disjunct as the argument, and returning TRUE as soon as one of the goal statements is satisfied.

Conjunction: select an evaluation order and map over the conjuncts, sending the SATISFY message to the RI with each conjunct as the argument; return TRUE only if all the conjuncts are satisfied. Any variable bindings which result from the satisfaction of a conjunct must be used for applied to subsequent conjuncts.

3. The RI sends the MATCH-P message to the goal pattern.

4. Upon receipt of the MATCH-P message, the goal pattern searches first searches the current working memory to determine whether the goal pattern matches any current data pattern objects. If a match is found, a binding object is created and a TRUE result is returned with the database pattern as the basis for the result.

5. If no match is found in the database, then the pattern object searches the current rule set, attempting to match the hypothesis components of rules. Rules yielding a successful match are collected together.

6. The Rule Interpreter selects one rule, based on a problem oriented conflict resolution criterion.

7. The Rule Interpreter sends itself a SATISFY message with the antecedent component of the rule as its argument. If a TRUE result is returned from this SATISFY message, then a TRUE result is also returned for the rule, with both the data pattern and the rule itself as the basis for the result.

At the end of this process, either the attempt to satisfy the goal will have failed, or it will have succeeded and the chain of inference from the data objects through the rules will be provided as the basis for the result.

4.2.3 Security Issues

Only a few special requirements are imposed on these processes by multilevel security, as we no'e here.

- The methods must support the ability to continue a search through lower level objects for appropriate rules and data
- There must be an ability to handle conflicts which arise when contradictory information or rules are stored at different access classes
- Newly created data must be stored in the database object at the access class of the process that creates the data

4.3 Inference Issues

Another set of issues arises when we consider correctness and consistency of inferences. In particular, since objects at high access classes may incorporate information which contradicts information at lower access classes, it is likely that high level inferences will, at times, contradict low level inferences. Furthermore, rules at high access classes may provide alternatives to rules at lower access classes. Selection of appropriate rules for achieving goals, or selection of which statements to believe becomes an important issue in a multilevel KBS.

These issues are addressed in work on *Truth Maintenance* in knowledge based systems [3]. A particular concern in truth maintenance systems (TMS) is handling default assumptions or beliefs that are later invalidated by new information. A common approach is to keep track of the basis for each belief and to use the basis in order to determine what to believe, once a contradiction occurs.

We advocate a similar process here. Low level data may be treated as default data by objects at higher access classes. When a contradiction between a low statement and a high inference is detected, a conflict resolution procedure must be invoked to determine which statement should be considered as representing the truth. The resolution of such conflicts must be determined on the basis of the information supporting each. Accordingly, we feel that it is essential to store with each information item the source of that information for use in conflict resolution. Interesting work on hierarchical nonmonotonic knowledge structures has been reported by Konolige [10]. We feel that this theoretical work is directly relevant to the issues raised here, and plan to investigate these connections in greater depth.

A similar problem arises from resolution of conflicts between rules to be used in order to satisfy a goal. We must have a basis for selecting the most appropriate rule to be used in each instance.

For maintaining a "correct" view of the world, we advocate the following conflict resolution rules.

- **Backward chaining:** When multiple rules are applicable, select the highest rule; if multiple rules still pertain, select on the basis of domain knowledge. This is based on the heuristic that the higher rule is more likely to be correct than the lower rule.
- **Forward chaining:** When a contradiction is generated, choose the statement generated by the highest level rule. This is based on the same heuristic as in the case of backward chaining.

Contradictions are of greatest concern when they occur within the same access class, since they may be more difficult to resolve. Contradictions between high and low statements should be less troublesome, presuming that "true" information is more likely to be sensitive and therefore placed in higher access classes than incorrect information. One possible heuristic for resolving such contradictions is to choose high information over low information. The presumption of information quality may not always be true, however, and in general contradictions must be resolved by examining the sources of the inconsistent statements in order to determine which should be accepted.

High statements that contradict low ones should not be a problem, since the high statements will be found first in a search through the working memory.

Handling contradictions implies the ability to recognize contradictions. Simple contradictions occur when a statement is inferred which is the negation of a statement already in the working memory. In these cases, a straightforward search through the database, possibly optimized through an intelligent indexing procedure, can identify contradictions and select the correct statement. In other cases, however, contradictions may exist when a set of statements are considered jointly. Uncovering these contradictions remains a difficult AI problem and is not considered here. If a general solution is found to the problem of implicit inconsistency among a set of statements, we believe that we will be able to adapt that approach for use in supporting multilevel security.

Chapter 5

Example

Here we illustrate a number of the issues raised in the preceding chapters. In particular, we will focus on the issue of correctness and contradiction, using a simple rule based system for illustration. We assume that the processes of achieving goals and generating inferences will be handled in an object oriented implementation as described above, and we will not address these issues here.

5.1 Scenario

Our examples are based on the following scenario. We have a secret mission that requires us to send a message to an operative working on a remote island in the Antarctic. The location of this operative is so sensitive that we must create an unclassified[1] scenario that will mask our true intent. Because of the distances involved and the uncertain Antarctic weather, our messenger must travel by sea. However, in order to conceal this fact at the unclassified level, we have indicated that our operative must travel by air. In the following example, we shall reason about which operative to send, Tweety or Opus. We follow the reasoning and show that our conclusions depend on the access class of the subject and the information available at

[1]The levels, unclassified and secret, are not intended to have any relation to US Department of Defense classifications, but were chosen on the basis of their descriptiveness.

each access class.

5.2 The Knowledge Base

Our information is represented by the following sets of rules and data items. In these examples, the construction $x is used to indicate the universal variable, x. We also use a prefix notation for predicates.

The set of rules available at the unclassified level, shown in Figure 5.1, represent the following knowledge:

- All birds fly.
- Something that is yellow is not black.
- All penguins can swim.
- All penguins are black.
- If something can fly and is small, then choose it. (This is our selection rule for choosing an operative.)

Our unclassified database, shown in Figure 5.2, represents the following facts:

- Opus is a bird.
- Tweety is a bird.
- Tweety is yellow.
- Tweety is small.

At the secret level, we have the additional rules shown in Figure 5.3. In particular, we know something more about penguins which was not known at the unclassified level. The rules at this level represent the following knowledge:

<u>**Unclassified Rules**</u>

Rule 1 If (BIRD $x)
  then (FLY $x)

Rule 2 If (YELLOW $x)
  then (NOT (BLACK $x))

Rule 3 If (PENGUIN $x)
  then (SWIM $x)

Rule 4 If (PENGUIN $x)
  then (BLACK $x)

Rule 5 If (FLY $x) AND (SMALL $x)
  then (CHOOSE $x)

...

Figure 5.1: The Unclassified Rule Set

<u>**Unclassified Data**</u>

(BIRD OPUS)
(BIRD TWEETY)
(YELLOW TWEETY)
(SMALL TWEETY)
...

Figure 5.2: Unclassified Data

Secret Rules

Rule 6 If (PENGUIN \$x)
    then (NOT (FLY \$x))

Rule 7 If (SWIM \$x) AND (BLACK \$x)
    then (CHOOSE \$x)

...

Figure 5.3: Classified Rules

Secret Data

(PENGUIN OPUS)
...

Figure 5.4: Classified Data

- No penguin can fly.
- If something is black and can swim, then choose it as the operative.

As Figure 5.4 shows, we have one additional fact at the secret level: we know that Opus is a penguin.

Now we wish to draw some conclusions from this data.

5.3 Choosing an Operative

We begin by trying to satisfy the goal, (CHOOSE \$Y). The value assigned to \$Y will be our choice, in the event of successfully achieving the goal. While we will gloss over many of the details, we will follow the general procedure outlined above for backward chaining.

We begin with the view from the unclassified level. The process is started when we send the message (SATISFY (CHOOSE \$Y)) to the Rule Interpreter. This goal is a single statement and is not already in the database.

Therefore, we next search for rules which could satisfy the goal. The single rule selected is Rule 5, with the variable $X bound to $Y. This results in the RI sending itself the message, (SATISFY ((FLY $Y) AND (SMALL $Y))). This is a conjunctive goal and requires successful solution of the two subgoals, (SATISFY (FLY $Y)) and (SATISFY (SMALL $Y)). The first subgoal matches rule 1, and results in the message (SATISFY (BIRD $Y)) being passed to the RI.

This subgoal may be satisfied in two ways, either by matching the statement (BIRD OPUS) or by matching (BIRD TWEETY). Assume that (BIRD OPUS) is matched first. This causes $X to be bound to OPUS, which modifies the subgoal, (SATISFY (SMALL $Y)) to be the more specific subgoal, (SATISFY (SMALL OPUS)). This subgoal ultimately fails, as there is no rule which is applicable, nor is there a statement in the database which matches the subgoal.

Here we take a back-tracking approach, and when a failure occurs, we return to the last point where a choice was made to try a new possibility. The last choice involved the match to (BIRD $Y), so now we try the second match, (BIRD TWEETY). This results in the more specific subgoal, (SATISFY (SMALL TWEETY)), being sent to the RI. This goal is satisfied by the statement in the database, (SMALL TWEETY). Each component of the conjunctive goal is now satisfied, and as a result, the conjunctive goal is also. This yields the final solution, (CHOOSE TWEETY). Tweety is chosen as the best operative for the job.

At the secret level, we find two rules are candidates for determining an operative, Rule 5 and Rule 7. Our conflict resolution process causes us to choose Rule 7, as it is the higher rule. The subgoals that result in this case are (SWIM $Y) and (BLACK $Y). The first subgoal does not match anything in the database, but does match the hypothesis component of unclassified Rule 3, resulting in the new subgoal, (PENGUIN $Y). This subgoal matches (PENGUIN OPUS), and binds $Y to OPUS. This result constrains the second subgoal of the conjunct to (BLACK OPUS). Again, this subgoal does not match anything in the database, but does match the Rule 4, resulting in the new subgoal, (PENGUIN OPUS), which is satisfied. The satisfaction of these subgoals leads to the satisfaction of the original goal, with OPUS as the resulting value.

Here, we have selected Opus as the best operative. By using information

<u>**Unclassified Data**</u>

(FLY OPUS)—(local inference: (BIRD OPUS) + Rule 1)
(FLY TWEETY)—(local inference: (BIRD TWEETY) + Rule 1)
(BIRD OPUS)
(BIRD TWEETY)
(YELLOW TWEETY)
(SMALL TWEETY)
...

Figure 5.5: Unclassified Data With New Inferences

privy to the secret level, we reach a different conclusion, just as we had desired.

5.4 Drawing Conclusions From Data

We now wish to consider forward-chaining operations or direct inference generation. There are many schemes for controlling such inference processes, which we will not concern ourselves with here. Instead, we will choose certain inferences for illustrative purposes.

First, we will consider inferences generated at the unclassified level. We begin by considering the statement, (BIRD OPUS). This statement matches the antecedent component of Rule 1, and therefore allows us to infer the statement, FLY OPUS. Similarly, we can infer (FLY TWEETY). We add both of these statements, along with their derivations, to the database, yielding the updated database shown in Figure 5.5. We could continue the process further, but we now wish to switch our attention to the secret level.

At the secret level, we generate the same inference that was generated at the unclassified level, (FLY OPUS), but with a different justification that the inference was made at the secret level. Only one secret statement, (PENGUIN OPUS), matches any rules. The matched rules are the unclassified rules 3 and 4, and the secret rule 6. These rules entail the new inferences, (SWIM OPUS), (BLACK OPUS), and (NOT (FLY OPUS)). Here, the only problem occurs when we attempt to add the last statement, (NOT

Secret Data

(NOT (FLY OPUS))–(local inference: (PENGUIN OPUS) + Rule 6)
(SWIM OPUS)–(local inference: (PENGUIN OPUS) + Rule 3)
(BLACK Opus)–(local inference: (PENGUIN OPUS) + Rule 4)
(BLACK Opus)
(PENGUIN OPUS)
...

Figure 5.6: Classified Data With New Inferences

(FLY OPUS)); here, we have created a contradiction. Using our conflict resolution rule, we accept the statement, (NOT (FLY OPUS)), and remove the statement (FLY OPUS), as the former statement was generated by a higher rule. The final database is shown in Figure 5.6.

We could continue the inference process, as we now have another rule which is applicable, but we will stop here. We have illustrated the point that different inferences could be generated at the different access classes. Within each access class, however, we are able to maintain a "correct" view, based on the information and knowledge available at the access class.

5.5 Discussion

In these examples, we have illustrated the control of inferencing processes within a multilevel environment. We were able to generate inferences appropriate to the knowledge available to users at different access classes, and to show how these inferences could be maintained separately, while providing users with information appropriate to their clearances.

The issue of correctness of inferences is an important one for any KBS. Correctness becomes particularly difficult when different information and knowledge is available to different actors in the process. However, we feel that multilevel security does not bring requirements which are significantly different or unique. Therefore, we feel that current methods under investigation by AI scientists for handling truth maintenance and nonmonotonic reasoning should be equally applicable here.

Chapter 6

Summary and Conclusions

6.1 Summary

In this report we have addressed two distinct issues involved with implementing a multilevel security policy for a KBS: an appropriate implementation medium and the correctness of inferences. We began by proposing a multilevel security policy for object oriented programming systems, based on the concept of multilevel objects. We identified several requirements for such a system and showed how our policy would support these requirements.

We next outlined an approach for implementing a particular form of KBS, a production system, within an object oriented framework. This allows us to ensure that multilevel security is maintained within such a system.

Finally, we identified a number of more subtle issues associated with maintaining the correctness of inferences within a multilevel security framework. These issues seem to be strongly linked to issues involved in nonmonotonic reasoning or truth maintenance in more standard knowledge based systems, and we propose methods similar to those used by truth-maintenance systems. In particular, we propose that the basis for an inference be stored with statements in the working memory. When a contradiction appears, the statement derived from the highest level information is the one to be selected.

This work has only begun to address the issues of multilevel security in KBSs, and while many important technical problems have only been identified at this time, we feel that the approaches we are following will lead to the ability to develop secure and habitable KBSs.

6.2 Comparison with the Single-Level Object Model

The composite object model we presented here is an extension of the single-level object model described by Millen and Lunt in their recent report [15]. Both models use an object oriented data model as a basis, and are very similar in many respects. In both models, objects have a single classification. However, in our composite object model, single-level objects can be combined into multilevel composite objects, where the component objects retain their identity as objects.

In both models, objects can contain references to other objects; one way an object can do this is through inheritance. Both models include a *hierarchy property*, which requires that the access class of an object must dominate that of its parent object.

Another way an object can refer to another object is by storing an object identifier (id) as the value of an instance variable. In the Millen-Lunt model, an object O_1 can contain a reference to another object O_2 only if the referenced object O_2 has an access class dominated by that of O_1 or if the referenced object O_2 was created by a subject whose access class is dominated by that of O_1. In contrast, in the composite object model presented here, if an object O_1 contains a reference to another object O_2, then the access class of O_1 must dominate that of O_2.

This difference means that, in the composite object model, the existence of an object is considered to be classified at the access class of the object itself. Thus, in this model, low users cannot know of the existence of high objects, and consequently low objects cannot contain references to high objects. In the Millen-Lunt model, however, low subjects may know of the existence of a high object if the high object was created by a low subject, and, by extension, of that class of high objects. Thus the object id of the high object can be stored as a value in a low object, if the high object was created by a low subject. The Millen-Lunt model hides the existence of a high object from low subjects if the high object was created by a high subject; in this case, the existence of a high object is protected by ensuring that its object id is not stored in a low object.

This brings us to the question of writing up. Neither model allows a subject to write into an object of a higher access class. However, the Millen-Lunt model allows low objects to create high objects. The composite object model, on the other hand, does not allow any writing up whatsoever. In particular, it does not allow low subjects to create high objects.

Neither model has polyinstantiation, although what Millen and Lunt call "apparent polyinstantiation" can arise in both models [15]. That is, object ids cannot be polyinstantiated because object ids are guaranteed to be unique. However, in the Millen-Lunt model, the user-defined object name or other identifying attribute can be polyinstantiated. Lunt and Millen called this "apparent polyinstantiation" in that polyinstantiation may "seem" to occur in the application, because the identifying attributes by which the user knows the instances may not be unique. For example, a low user can create an instance of a low "employee" object with the same employee name as an invisible high instance of "employee." This polyinstantiation is only apparent, because the two instances have unique and distinct object ids. But to the high user there appear to be two occurrences of the same employee at different access classes.

The composite object model also allows apparent polyinstantiation, but in all cases such polyinstantiation occurs as the result of a decision to polyinstantiate by a high subject. Unlike the Millen-Lunt model, polyinstantiation cannot occur simply because some action by a low user results in a name conflict, as in the above example. We sometimes call this "shadowing" to distinguish it in our model.

The Millen-Lunt single-level object model explicitly models integrity constraints and classification constraints, whereas the composite object model does not. However, these constraints can be easily incorporated into the composite object model framework in much the same way as for the Millen-Lunt model.

The Millen-Lunt single-level object model shows how inference rules can be represented within the model framework. The composite model takes this idea much further.

Both models rely on an underlying security kernel for the enforcement of mandatory security. This means that with both models, the system layer providing the object oriented interface can be untrusted with respect to mandatory security. Neither model addresses discretionary security.

Other differences between the models are as follows. The Millen-Lunt model carefully defines the role of subjects and contains several properties that govern the activity of subjects. By contrast, the composite model presented here does not in any way constrain the implementation of subjects. The Millen-Lunt model describes a set of system operations and discusses the security requirements and constraints relevant to those operations. The composite object model does not define a list of operations. The Millen-Lunt model includes some detail on how such a system might actually be implemented. This composite object model does not.

6.3 Conclusion

In this project we have developed three distinct security policy models, the Multilevel Object Model, the Single-Level Object Model, and the Composite Object Model. We focused primarily on the latter two models.

In this report we describe the Composite Object Model which builds on the foundation of the Millen and Lunt [15] single-level object model in order to develop a model capable of representing multilevel entities. We have taken an approach in which multilevel objects can be constructed from single-level components, and our approach avoids the difficulties of earlier models of multilevel objects [12]. We feel that the real-world applications that may make use of a multilevel KBS will require the ability to model multilevel objects directly.

As in the work by Millen and Lunt, we have shown here how a multilevel KBS can be implemented by using a multilevel object oriented database system as a framework. We have shown how such a framework can be modeled to provide the ability to represent multilevel objects, and we have shown how realistic reasoning systems can make use of such a framework. We have shown that the multilevel object oriented model we present here can be implemented as an untrusted system layer on a conventional reference monitor that enforces mandatory security. This is important for several reasons. First, it demonstrates the feasibility of building a high-assurance multilevel KBS. The "trust" in trusted computing systems rests on the ability to provide convincing arguments or proofs that the security mechanisms work as advertised and cannot be disabled or subverted. In building a multilevel KBS, providing such assurance is potentially problematical because of the size and complexity of the system. Achieving Class A1 assurance is possible only if that portion of the system enforcing mandatory security is small and isolated. Building on a previously verified reference monitor for mandatory security satisfies this requirement.

Secondly, building the KBS as an untrusted system layer on a conventional reference monitor means that such systems can be constructed in a cost-effective manner, reusing established security solutions, and avoiding the need for a large verification effort. avoiding the need for a large verification effort.

Third, such an architecture suggests the possibility that multilevel KBS can be built to provide the full functionality users expect from such systems. The architecture we envision here is similar to that of SeaView, in that multilevel composite objects are broken into single-level objects that can in turn be stored in storage objects of the corresponding access class and managed by an underlying reference monitor [14]. Both designs include the basic functionality required by multilevel applications utilizing an integrated collection of information classified at different security levels.

Fourth, this approach provides the greatest degree of security possible, because it considerably reduces the disclosure risk to sensitive information. This is because the KBS is governed by the underlying reference monitor, which partitions data according to their classification. Thus, no subject can gain access to any information whose classification is not dominated by the subject's access class. All operations can be handled by single-level untrusted subjects. This is the most conservative approach possible for mandatory security. Any other approach would require the use of trusted subjects, which would admit additional risk.

Finally, a system constructed in this way can take advantage of the TCB subsetting evaluation strategy [18, 11].

However, there are still many implementation issues to resolve if we are to construct a multilevel KBS in this way. It is almost certainly the case that current reasoning systems have not been implemented such that it would be straightforward to port them to run on a reference monitor. The TCB subsetting approach means that there must be at least one instance of the system for each active security level (a security level is considered to be active if there is a subject active at that level). Thus, the KBS must be able to support multiple system instances that share the same logical knowledge/data base. Conventional concurrency control and recovery mechanisms will not work in a multilevel environment. Global data structures cannot be used. Thus, multilevel security will have a significant impact on the design of a KBS because the design will be constrained by the underlying reference monitor.

Although we believe that the work presented here is a significant advance in the state of the art, further work is necessary to prove some of the concepts we have introduced. Construction of a prototype system would expose the implementation issues and prove the viability of the approach. In addition, further work is necessary to identify and model the discretionary security issues for KBSs.

6.4 Future Directions

In this report we have presented a model for a multilevel KBS. Although the work presented here is a significant first step, further work is necessary to prove some of the concepts we have introduced and to evaluate the overall habitability of the system. In particular, the issue of truth maintenance methods and their effects on the overall usability of the system needs to be investigated. The next step would be to build a proof-of-concept prototype system to prove the viability of our approach. We see two alternatives for developing a prototype system.

- The prototype could be constructed to run on a high-assurance reference monitor, such as the Gemini Computers GEMSOS system. To do this, a rudimentary reasoning system would have to be built from scratch to run on the reference monitor. The resulting prototype would have only very limited functionality, but would be multilevel secure and would demonstrate the feasibility of the architectural approach.

- The prototype KBS could be constructed on a single-level OOPS and built so as to mimic the effects of multilevel security. Although the resulting prototype would not in fact be multilevel secure, it would be able to demonstrate the effects of multilevel security on the applications that use the KBS.

In addition to building a proof-of-concept prototype, it would be profitable to develop a sample application for implementation on the prototype. This task would best be undertaken with the second approach to building the prototype. The advantage of developing a sample application would be to test the adequacy of the model for realistic applications and to identify potential strategies for realistic applications and to identify potential strategies for dealing with such complications as shadowing and the design of multilevel composite objects.

It is important that the subset of the knowledge base visible at any access class be capable of yielding meaningful results. It is also important that the subset of the knowledge base visible at any access class be closed with respect to inference of higher level subsets. Developing a knowledge base that satisfies these requirements will be a challenging new task. Tools will be required to assess the impact of specific sets of data/knowledge classifications. Because its knowledge is encoded for logical processing, a KBS is potentially able to reason about the closure of subsets of its knowledge base with respect to inference. Methods have been proposed for using AI techniques for locating potential inferences arising from classification inconsistencies in relational databases [5]; analogous techniques may be applicable for multilevel knowledge bases. For a KBS, such reasoning may be able to take advantage of the fact that the much of the semantics of the application is captured in the knowledge base.

Discretionary access control is also relevant to a KBS and deserves attention in future work. A KBS has several distinct types of users, each with their own set of authorized capabilities. Because knowledge changes frequently, there are many types of engineer and expert with roles to play in maintaining the knowledge base. There may be a czar to resolve conflicts when the experts disagree. There will be test engineers who develop and run test cases on proposed modifications to the knowledge base. Such users may be restricted to certain subsets of the knowledge base, and these subsets may be appropriate objects of discretionary access control. Finally, there are the consumers, users of the system, for whom discretionary access controls seem especially relevant. Further study is necessary to examine appropriate discretionary security policies for KBSs.

The security model presented here could also be extended to model integrity properties as well as security properties.

Future work could formalize the model we have presented here and specify a set of primitive operations that could be formally verified to meet the basic properties of the model.

In conclusion, we believe that we have defined a workable and verifiable security policy for both OOPS and KBS built on top of them. The next step is to implement a prototype system and evaluate issues relating to habitability.

Bibliography

[1] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, Bedford, MA, March 1976.

[2] E. Charniak and D. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, Reading, MA, 1985.

[3] J. deKleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[4] D. E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, MA, 1982.

[5] D. E. Denning and M. Morgenstern. Military database technology study: AI techniques for security and reliability. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1986.

[6] *Department of Defense Trusted Computer System Evaluation Criteria, DOD 5200.28-STD.* Department of Defense, December 1985.

[7] A. Downing, I. Greenberg, and T. F. Lunt. Issues in distributed database security. In *Proceedings of the 5th Aerospace Computer Security Conference*, December 1989.

[8] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

[9] K. Konolige. Bayesian methods for updating probabilities. In R. O. Duda, P. E. Hart, K. Konolige, and R. Reboh, editors, *A Computer-Based Consultant for Mineral Exploration*, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, 1979.

[10] K. Konolige. Lecture notes in artificial intelligence. In *Non-Monotonic Reasoning*. Springer-Verlag, June 1988.

[11] T. F. Lunt. Multilevel database systems: Meeting class A1. In C. E. Landwehr, editor, *Database Security II: Status and Prospects*. North Holland, 1989.

[12] T. F. Lunt. Multilevel security for object-oriented database systems. In D. L. Spooner and C. E. Landwehr, editors, *Database Security III: Status and Prospects*. Elsevier, 1990.

[13] T. F. Lunt, D. E. Denning, R. R. Schell, W. R. Shockley, and M. Heckman. The SeaView security model. *IEEE Transactions on Software Engineering*, June 1990.

[14] T. F. Lunt, R. R. Schell, W. R. Shockley, M. Heckman, and D. Warren. A near-term design for the SeaView multilevel database system. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, April 1988.

[15] J. K. Millen and T. F. Lunt. Secure knowledge-based systems. Technical Report SRI-CSL-90-04, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1989.

[16] M. Minsky. A framework for representing knowledge. In *The Psychology of Computer Vision*. McGraw-Hill, New York, 1975.

[17] G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, Princeton, NJ, 1976.

[18] W. R. Shockley and R. R. Schell. TCB subsetting for incremental evaluation. In *Proceedings of the Third AIAA Conference on Computer Security*, December 1987.

[19] G. W. Smith. Identifying and representing the security semantics of an application. In *Proceedings of the Fourth Aerospace Computer Security Applications Conference*, December 1988.

[20] G. L. Steele Jr. *Common LISP*. Digital Press, 1984.